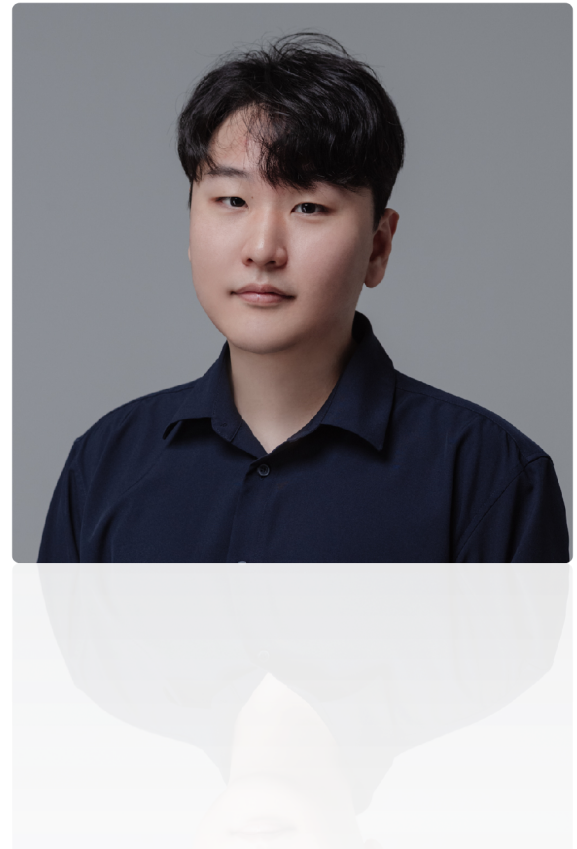
K DEVCON

소프트웨어 파괴의 미학

이선협 / @kciter

발표자 소개

- 이선협 (@kciter)
- CoBALT, CTO
- https://github.com/kciter
- https://kciter.so



불확실성에 대항하기

이런 고민을 해본 적 있나요?

소프트웨어 개발은 너무 많은 것이 애매모호하다

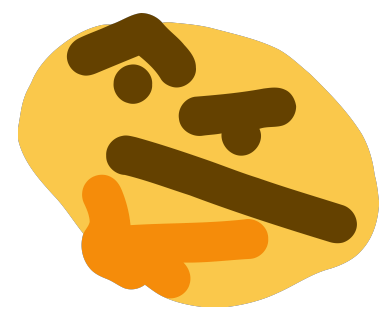
항상 나를 괴롭히던 고민

- 코딩으로 밥벌이를 한지 벌써 10년이 넘었지만…
  - 어째서 아직도 무언가를 **완성하는 것**이 어렵게 느껴지는가?
  - 아직 지식과 경험이 부족한 것인가?
- 애매모호함 앞에서 겸손해지는 나…

아주 작은 깨달음

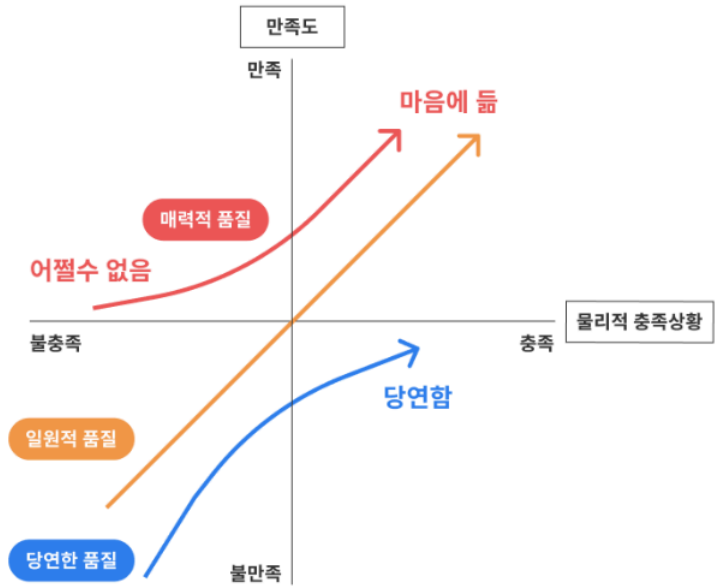
소프트웨어 개발이 정말 어려운 이유는 **많은 것이 불확실**하기 때문

무엇이 소프트웨어 개발을 불확실하게 만드는걸까?



큰 이유 - 비즈니스라는 괴물

- 아무리 잘 설계해도 비즈니스는 급변한다.
  - 가설은 언제나 틀릴 수 있다.
  - 요구사항이 변경되는 것은 당연하다.
  - 경쟁자는 기다려주지 않는다.
  - 도메인 지식 또한 계속 변한다.
- 따라서 개발자의 예측은 높은 확률로 틀릴 수밖에 없다.
  - 확장에 대한 대응이 무의미해질 수 있다.
  - 아키텍처가 적합하지 않아질 수 있다.
  - 공든 탑이 무너지는 슬픔…
  - 내가 작성한 코드는 순식간에 **기술 부채**가 된다.



카노 모델

시간이 지나도 비즈니스의 복잡성은 죽지 않는다

당시에 완벽했던 코드도 시간이 지나면 가치가 없어진다

작은 이유 - 지식과 경험의 부족

- 우리는 왜 리팩터링을 하는가?
  - 코드를 이해하기 힘들어서
  - 성능에 문제가 있어서
  - 확장하기 힘들어서

작은 이유 - 지식과 경험의 부족

- 우리는 왜 리팩터링을 하는가?
  - 코드를 이해하기 힘들어서
  - 성능에 문제가 있어서
  - 확장하기 힘들어서
- 리팩터링은 **지식과 경험에 기반한 기술**
  - 리팩터링을 하기에 **가장 좋은 시기는 어떻게 리팩터링을 해야할 지 정확히 알 때**
  - 적절한 기술을 사용할 수 없다면 올바르게 고치는 것은 불가능
    - 다시 작성해도 다시 리팩터링을 하게 될 수 있다.
- 리팩터링이란 지식과 경험이 부족했기에 해야하는 **개발자 스스로 만들어낸 기술부채**

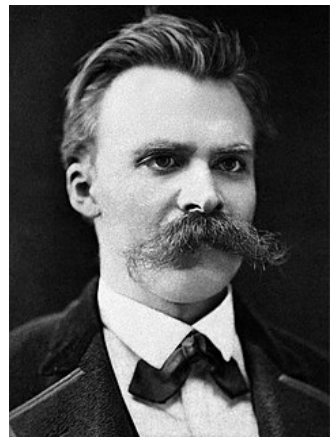
극복을 위한 노력

- 수학적인 모델링
- 설계 패턴 적용
- 방법론에 대한 공부
- 비즈니스에 대한 이해
- …

결국 은총알은 없는가?

은총알은 없다

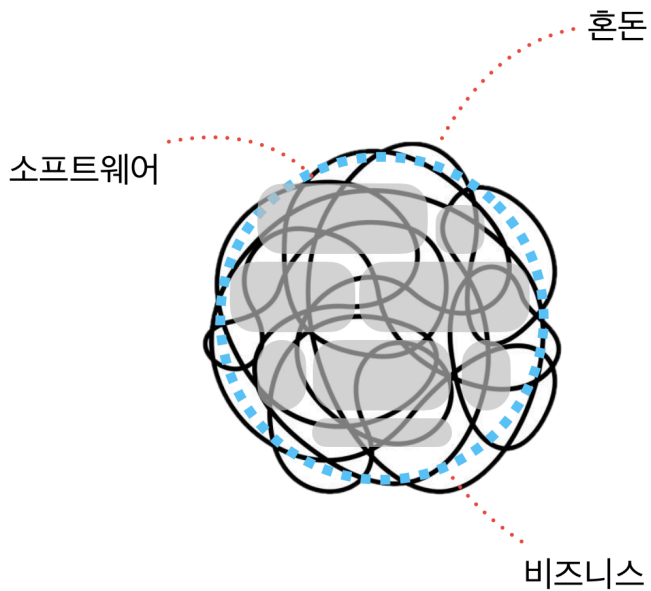
- 모든 것은 Trade-off라는 것을 배우게 된다.
- 따라서 완벽이라는 것은 없다는 것을 깨닫는다.
- 완벽할 수 없다면 확신할 수 없다.
- 우리는 언젠가 **삭제할 수 밖에 없는 코드**를 만들게 된다

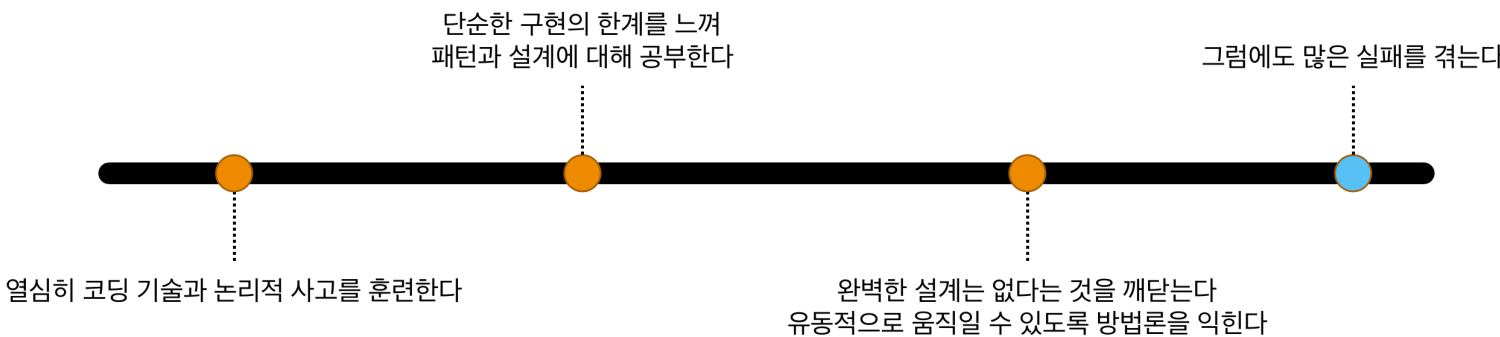


프리드리히 니체

비극의 탄생

- **경계를 파괴하는 힘**
  - 현실, 자연, 무질서, 파괴, 죽음
  - 디오니소스적인 것
- **경계를 인식하고 나누는 힘**
  - 디오니소스적인 힘에 대항하는 이성적인 방법들
  - 분석, 분류, 검증
  - 아폴론적인 것
  - 대부분의 개발자는 매우 아폴론적인 존재


혼돈
소프트웨어
비즈니스


단순한 구현의 한계를 느껴
패턴과 설계에 대해 공부한다
그럼에도 많은 실패를 겪는다
열심히 코딩 기술과 논리적 사고를 훈련한다
완벽한 설계는 없다는 것을 깨닫는다
유동적으로 움직일 수 있도록 방법론을 익힌다

기술만으로는 해결할 수 없는 문제

여기서 개발자는 두 가지 선택을 할 수 있다

염세주의적 태도

- 디오니소스적인 힘에 대항하는 것은 **무의미**하다
  - 어차피 바뀔거야 대충 만들자
  - 설계, 방법론같은 것은 무의미하다
  - 애자일? 제대로 돌아가는 걸 본 적이 없다
- 과연 이것이 옳은 태도일까?
  - 다들 안좋다고 생각하겠지만 실제로는 많은 개발자가 염세주의적 태도를 취한다.
  - **편하고 안락한 길**

맞서 싸우기

- 디오니소스적인 힘을 **받아들이는 것**
  - 완벽할 수 없다는 것을 깨달았다
  - 그럼에도 불구하고 나아가야 한다
- 어떻게 극복해야 하는가?
  - 다음 길을 개척하고 발굴해야 한다
  - **어렵고 견뎌야 하는 길**

혼돈에 대항한 소프트웨어 업계의 역사

- 추상화 없는 계산의 수행
- 제어 구조의 발명
- 데이터 추상화
- 프로그래밍 패러다임
- MDA
- 재사용 가능한 패턴
- 애자일
- DDD
- 리팩터링 기술
- …

이제 어떻게 나아가야 할까?

제안

- '어차피 지워진다'라는 염세주의적 생각에 빠지기 쉬운 것이 현실
- 그렇다면 거꾸로 생각해서 차라리 잘 지울 수 있게 만들면 어떨까?
- 즉, **파괴**에는 **파괴**로 대응해보자!

파괴 주도 개발

파괴는 안좋은 것일까?

새로운
무엇인가는
구조마저
의심하고
파괴하지
않고서는
태어나지 않아!

소프트웨어를 파괴하는 두 가지

- 기능을 삭제하는 것 / Pivoting
  - 피벗은 제품이 더 좋은 길로 나아갈 수 있는 **기회**
- 같은 기능을 다시 만드는 것 / Refactoring
  - 리팩터링은 소프트웨어의 **생명을 연장**시키는 일
  - 리팩터링이 필요한 이유는 **경제성**
  - 많은 기능을 빠르게 추가하기 위함

다행히 나 혼자만의 생각은 아님

Greg Young - The Art of Destroying Software

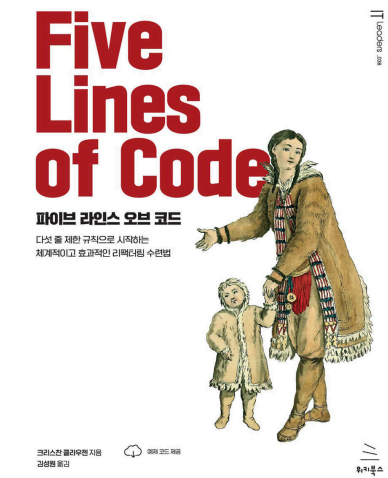
- Big ball of mud paper. It argues that the big ball of mud is inevitable.

  The way to tackle it is to create pockets of smaller ball of muds. You have to write code for the purpose of deleting it.
- Make your system easier to delete than to change
- Don’t plan for future changes, focus on the ability to rewrite from scratch when changes happen.
- https://www.youtube.com/embed/1FPsJ-if2RU?si=00mBh99JDY7EMgi7

Five Lines of Code

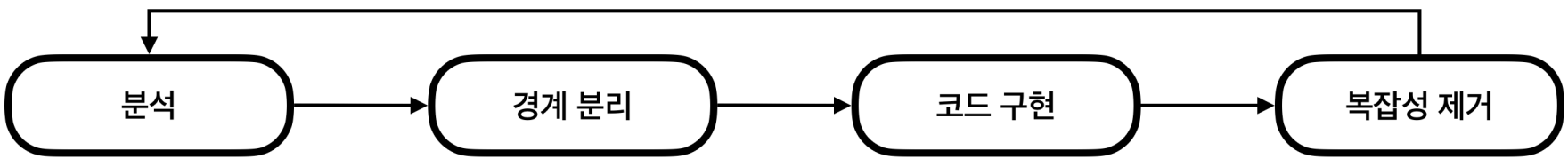
- 시스템에는 어느 정도 오래 지속되는 코드가 필요합니다. 도메인의 복잡성에 따라 그 필요한 양은 다릅니다.

  그러나 이번 장에서 단 한 가지만 말하라면 바로 '적은 것이 더 낫다'입니다.
- 9.1. 다음 시대는 코드를 지우는 시대일 것이다
- 우리는 아직 코드 삭제에 익숙하지 않습니다. 나는 이것이 다음에 해결해야 할 큰 과제라고 생각합니다.



파괴 지향 개발?

- 언젠가 코드가 파괴될 것이라는 사실을 받아들이고, 그것을 지향하여 개발하는 방법론
- 세 원칙
  - 불확실성이 있는 것과 없는 것을 파악한다.
  - 여러 방법을 선택할 수 있다면 파괴하기 쉬운 쪽을 선택한다.
  - 필요한 것만을 유지한다. 따라서 필요 없는 것은 전부 지운다.
- 복잡성을 인지할 수 있는 것과 아닌 것으로 구분하는 것이 중요
- 아직 발전 중


분석
경계 분리
코드 구현
복잡성 제거

경계 분리

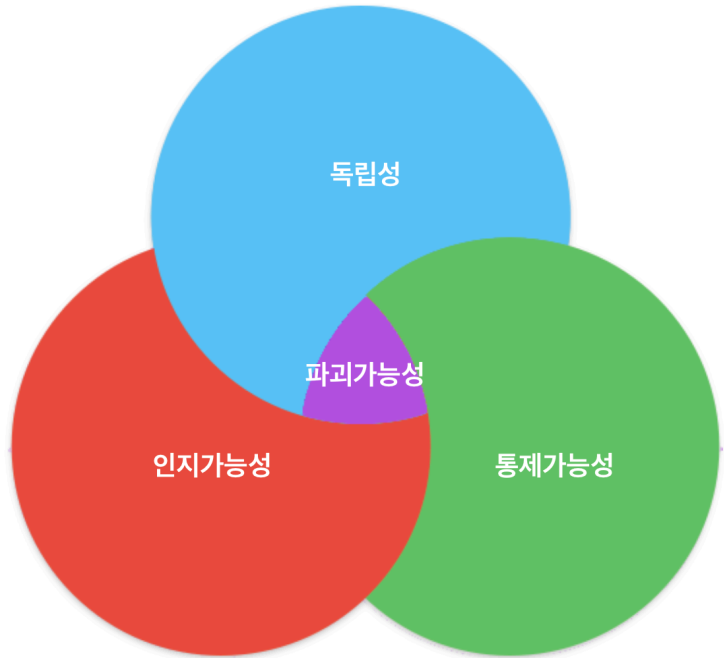
- 불확실성이 높은 것과 낮은 것을 **분리**하는 것이 중요
- **변화율 = 불확실성**
- 변화율을 측정하는 방법
  - 기능이 실험적인가?
  - 기능이 릴리즈 되었는가?
  - 고객의 반응이 어떤가?
  - 기능이 릴리즈된 후 얼마나 지났는가?
  - 기능이 복잡한가?
  - 적절한 지식과 경험이 있는 상태에서 만들었는가?
  - 코드에 신뢰성이 있는가?
  - 코드를 이해할 수 있는가?
  - 목표 성능을 달성했는가?

무엇부터 분리할까?

- 분리 단위
  - 애플리케이션, 모듈, 유즈케이스, 클래스, 컴포넌트, 메서드 등
  - 무엇을 기준으로 분리할 것인지 **추상화 레벨**을 결정해야 한다.
- 복잡한 비즈니스 로직일 수록 변화율이 높다.
- 기능은 시간이 지날 수록 변화율이 높아진다.
- 지원 서브 도메인에 가까울 수록 변화율이 낮다.
  - 비즈니스를 지원하는 활동
  - 계정 관리, 이메일 발송, 로깅, 모니터링 등
- 독립적으로 존재할 수 있다면 분리할 수 있다.
  - 인터페이스 만으로 통신할 수 있게 만들어야 한다.
- 불확실성을 측정할 수 없다면 억지로 분리하지 않는다.
  - 큰 단위에선 분리하지 않더라도 불확실성을 판단할 수 있는 추상화 단위에서 분리한다.

파괴하기 쉽게 만들기

- Make the change easy, then make the easy change - Kent Beck
- 파괴하기 쉽다는 것은 어떻게 측정하는가?


독립성
파괴가능성
인지가능성
통제가능성

파괴 가능성

- 독립성
  - 결합도와 응집도
  - 단일 책임 원칙
- 인지가능성
  - 코드를 인지할 수 있는가?
  - 코드의 가독성
- 통제가능성
  - 내가 통제할 수 있는 영역인가?
  - 코드오너쉽
  - 코드의 신뢰성
  - 코드의 사회성

복잡성 제거

- 사용하지 않는다면 코드베이스에 지워라
  - 추억과 애정이 담긴 코드여도 예외는 없다
  - T) 어차피 커밋 로그에 남아있으니까 복구 가능~
- 파괴 가능성이 낮은 코드는 리팩터링하거나 다시 만들어라
  - 시간이 부족하거나 불확실성이 높다면 주석을 남긴 후 나중에 처리한다.
- **최대한 단순성을 유지하는 것이 핵심**

Talk is cheap. Show me the code.
— Linus Torvalds —

코드 파괴의 기술

어떤 코드가 파괴하기 쉬울까?

- 변화율 기록하기
- 코드 가독성 끌어올리기
- 단계 분리하기
- 참조 투명성
- 단일 책임 원칙
- 인터페이스 분리 원칙
- 스트랭글러 무화과 패턴
- 메서드 전문화
- 중복 코드 작성
- **확장을 대비하기 보다 파괴에 대비한다라는 사고 방식이 중요**

변화율 기록하기

- 변화율이 높을 것다고 예상되는 코드엔 **주석**을 남긴다.
- 추후 정리하거나 다른 개발자가 확인할 때 편하다.

코드 가독성 끌어올리기

- 성능 이슈가 없다면 선형적, 선언적으로 작성해야 한다.
- 선형적이고 선언적일 수록 가독성이 높아진다.
- 가독성이 낮으면 불안감에 삭제하기 힘들어진다.

```
// 위에서 아래로 한 번만 읽으면 된다
fun binaryToDecimal(input: String): Int {
  return input
    .reversed()
    .mapIndexed { index, char →
      if (char == '1') {
        2.0.pow(index).toInt()
      } else {
        0
      }
    }
    .sum()
}
```

```
// 루프를 머리에서 연산해야 한다
fun binaryToDecimal(input: String): Int {
  var decimal = 0
  var binary = input
  var power = 0

  while (binary.isNotEmpty()) {
    val lastChar = binary.last()
    binary = binary.dropLast(1)
    if (lastChar == '1') {
      decimal += 2.0.pow(power).toInt()
    }
    power += 1
  }

  return decimal
}
```

🟢

```
// 함수명에서 의도를 바로 파악할 수 있다.
fun generateSequentialNumber(count: Int): Sequence<Int> {
  return generateSequence(1) { it + 1 }
    .take(count)
}

fun main() {
  print(
    generateSequentialNumber(10)
      .filterOdd()
      .sum()
  )
}
```

🟡

```
// 내용을 한 번 더 읽어야 한다.
fun main() {
  print(
    generateSequence(1) { it + 1 }
      .take(10)
      .filter { it % 2 == 1 }
      .fold(0) { a, b -> a + b }
  )
}
```

단계 분리하기

- 주로 프로그래밍 로직은 대체로 단계로 나눠서 작성하는 것이 가능
  - 대부분의 로직이 전처리, 계산, 후처리와 같은 형태를 지닌다.
- 단계를 분리하면 필요한 부분만 재작성이 가능하다.

```kotlin
// 전처리, 계산, 후처리가 분리되어 있다
fun makeFibonacciList(n: Int): List<Int> {
  val list = mutableListOf(0, 1)
  for (i in 2 until n) {
    list.add(list[i - 2] + list[i - 1])
  }
  return list
}

fun listSum(list: List<Int>): Int {
  return list.sum()
}

fun main() {
  val list = makeFibonacciList(10) // 데이터 생성
  val sum = list.sum() // 로직
  print(sum) // 출력
}
```

```kotlin
// 전처리, 계산, 후처리가 섞여있다
fun fibonacciSum(n: Int): Int {
  var sum = 0
  var a = 0
  var b = 1
  var c = 0
  for (i in 0 until n) {
    // 데이터 생성과 동시에 계산
    if (i ⩽ 1) {
      sum += i
    } else {
      c = a + b
      sum += c
      a = b
      b = c
    }
  }
  return sum
}

fun main() {
  print(fibonacciSum(10)) // 출력
}
```

참조 투명성 보장

- 같은 입력에 대해 항상 같은 출력을 보장하는 것
- 순수 함수, 불변 객체 등
- **코드 신뢰성**과 관련이 있다.
- 만약 지켜지지 않았는데 코드가 잘 작동한다면…? 😱
  - 영향을 미치는 모든 곳이 불확실해진다.
  - 함부로 건들면 결함이 생길지도 모른다는 불안감이 생겨 삭제하기 힘들어진다.
- 최소한 멱등성이라도 지녀야 한다.

🟢

```
// 순수 함수
fun factorial(n: Int): Int {
  if (n == 0) {
    return 1
  }
  return n * factorial(n - 1)
}
```

```
// 사이드 이펙트가 있지만 멱등성을 지닌 함수
val cache = mutableMapOf<Int, Int>()
fun factorial(n: Int): Int {
  if (n == 0) {
    return 1
  } else if (cache.containsKey(n)) {
    return cache[n]!!
  }
  val result = n * factorial(n - 1)
  cache[n] = result
  return result
}
```

🔴

```
// 참조 투명성이 없는 함수
var result = 1
fun factorial(n: Int): Int {
  if (n == 0) {
    return result
  }
  result *= n
  return factorial(n - 1)
}

// 나중에 result 변수가 변경된다면..?
```

단일 책임 원칙

- 가장 이해하기 쉬우면서 가장 적용하기 어려운 원칙
- **단일 책임의 기준**이 무엇인가?
- 요구사항을 잘 이해할 필요가 있음

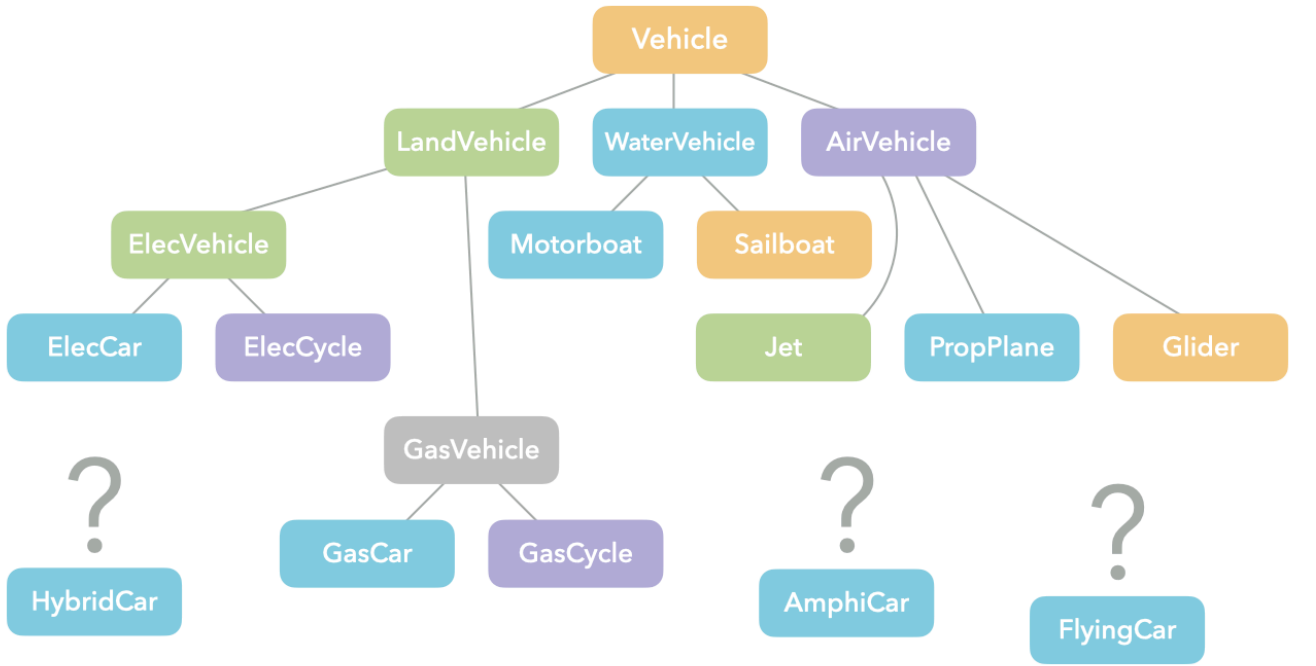
다음 코드는 단일 책임 원칙을 지키는가?

```
class UserInfo {
  val userId: Long
  val userName: String
  val email: String
  val telephone: String
  val provinceAddress: String // 도
  val cityAddress: String // 시
  val regionAddress: String // 구
  val detailAddress: String // 상세 주소
  val avatarUrl: String
  val createdAt: OffsetDateTime
  val updatedAt: OffsetDateTime
  val lastLoginAt: OffsetDateTime
}
```

- 두 가지 견해가 있을 수 있다.
  - 🙆‍♀️ 사용자와 관련된 정보가 포함되어 있고 모든 속성과 메서드가 사용자와 같은 비즈니스 모델에 속해 있어 만족한다.
  - 🙅‍♂️ 주소 정보와 시간 정보에 대한 비율이 상대적으로 크기 때문에 UserAddress로 분리할 수 있어 책임을 분리할 수 있다.
  - 요구사항에 따라 다르다!

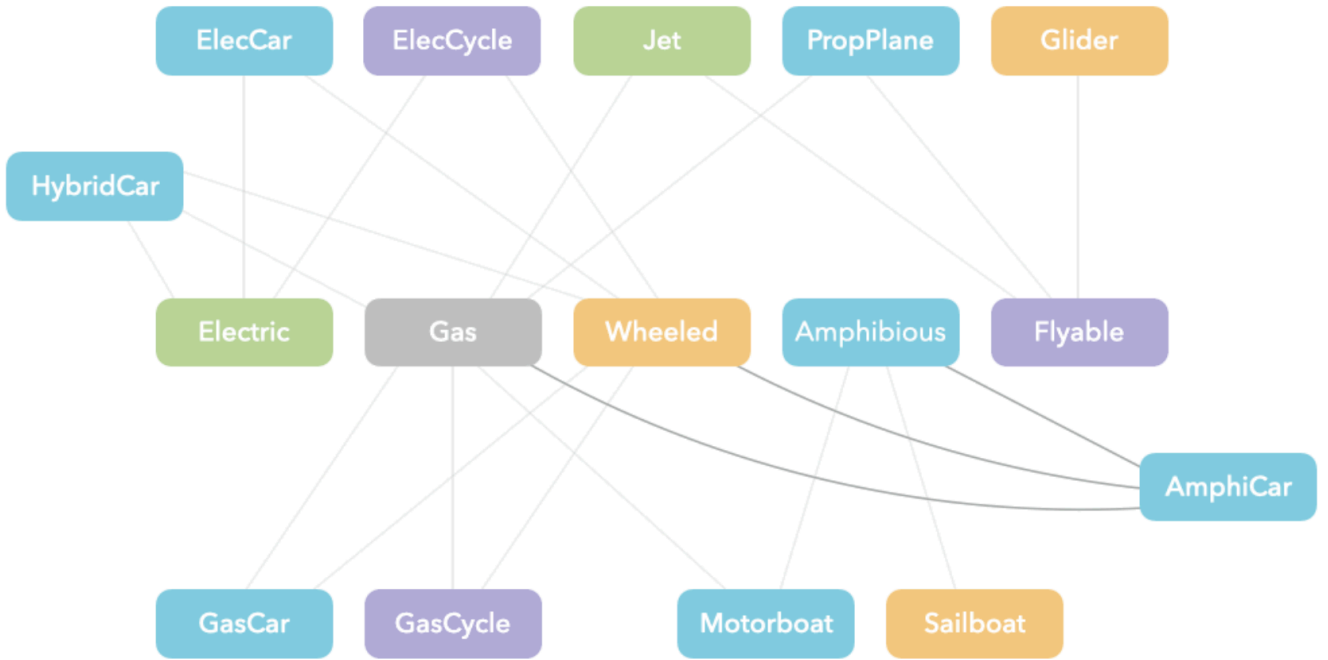
인터페이스 분리 원칙

- 클라이언트는 필요하지 않은 인터페이스에 의존하면 안된다.
  - 클라이언트는 인터페이스 호출자
- 인터페이스 계의 단일 책임 원칙
- **기능을 제거할 필요가 있다면 인터페이스를 지우기만 하면 된다.**



bit.ly/protocol-oriented-programming

삭제하기 힘든 설계



bit.ly/protocol-oriented-programming

만약 인터페이스를 분리해서 만든다면?

```
interface Bird {
  fun fly()
  fun walk()
  fun swim()
}

class Sparrow : Bird {
  override fun fly() = //  ...
  override fun walk() = //  ...
  override fun swim() = //  ...
}

class Penguin : Bird {
  override fun fly() = //  ...
  override fun walk() = //  ...
  override fun swim() = //  ...
}

class Ostrich : Bird {
  override fun fly() = //  ...
  override fun walk() = //  ...
  override fun swim() = //  ...
}
```

```
interface Flyable {
  fun fly()
}

interface Walkable {
  fun walk()
}

interface Swimmable {
  fun swim()
}
```

```
class Sparrow : Flyable, Walkable {
  override fun fly() = //  ...
  override fun walk() = //  ...
}

class Penguin : Walkable, Swimmable {
  override fun walk() = //  ...
  override fun swim() = //  ...
}

class Ostrich : Walkable {
  override fun walk() = //  ...
}
```

잘 작동하는 코드 교체하기

- 스트랭글러 무화과 패턴 (Strangler pig pattern)
  - https://martinfowler.com/bliki/StranglerFigApplication.html
- 레거시 코드를 교체할 때 사용하는 패턴
  - 레거시 코드 일부를 새로운 코드로 교체
  - 안정화 기간이 지난 후 대체된 레거시 코드를 제거
  - 반복
- **기존 코드를 교체하는 것이 아니라 새로운 코드를 추가하고 기존 코드를 감싸는 방식**

스트랭글러 무화과 패턴

Step 1

```
class LegacyCode {
  fun oldMethod1() {
    // ...
  }

  fun oldMethod2() {
    // ...
  }
}
```

스트랭글러 무화과 패턴

Step 2

```
class LegacyCode {
  fun oldMethod1() {
    // ...
  }

  fun oldMethod2() {
    // ...
  }
}

class NewCode {
  private val legacyCode = LegacyCode()

  fun newMethod1() {
    // ...
  }

  fun oldMethod2() {
    legacyCode.oldMethod()
  }
}
```

스트랭글러 무화과 패턴

Step 3

```
class NewCode {
  fun newMethod1() {
    // ...
  }

  fun newMethod2() {
    // ...
  }
}
```

스트랭글러 무화과 패턴

Step 2

```
class LegacyCode {
  fun oldMethod1() {
    // ...
  }

  fun oldMethod2() {
    // ...
  }
}

class NewCode {
  private val legacyCode = LegacyCode()

  fun newMethod1() {
    // ...
  }

  fun oldMethod2() {
    legacyCode.oldMethod()
  }
}
```

메서드 전문화

- 메서드의 일반화를 줄이고 더 특정한 목적을 가지도록 하는 것
- 개발자의 본능을 거스르는 일
- 분리되는 만큼 삭제하기 편하다.

메서드 전문화

Step 1

```
fun udpateUser(userId: Long, email: String, telephone: String, address: Address) {
  // ...
}
```

메서드 전문화

Step 2

```
fun updateUserEmail(userId: Long, email: String) {
  // ...
}

fun updateUserTelephone(userId: Long, telephone: String) {
  // ...
}

// ...
```

중복 코드 작성

- 변화율이 높은 곳에선 일부러 중복 코드를 작성한다.
  - 둘 중 하나는 분명 변경될 것이라는 확신이 있을 때
  - 개발자의 경험과 직관에 따라 선택
- 개발자의 본능을 매우 거스르는 일… 😱
  - 따라서 예제 코드는 작성하지 않겠다…

마치며

이미 있는 것을 잘 섞은 것 아닌가요?

맞습니다

하지만 바라보는 관점이 다릅니다

무엇이 중요한가?

- 잘 작성한 코드는 파괴하기도 쉽다.
- 개발자 내면의 본능을 잠시 억누르자
  - 경계를 나누는 것, 확장하는 것, 코드에 대한 애정…
- 중요한 것은 **좋은 소프트웨어를 만드는 것**
- 염세주의는 소프트웨어의 **독**이다.
- 끊임없이 **무엇이 좋은가**를 생각하자

DEVCON K

감사합니다

이선협 / @kciter